BID – Binary-Integer Decimal Encoding for Decimal Floating Point

A Format Friendly to Software Emulation and Compiler Native Support

Ping Tak Peter Tang
Intel Corporation
peter.tang@intel.com

Original: June 17, 2005
Last Updated: July 12, 2005
Minor Modifications: January 3, 2006

1 Summary

The current IEEE 754R proposal on decimal floating point has two components: arithmetic behavior and datatype encoding. The proposed arithmetic behavior is consistent with the need for realistic decimal processing; and the encoding is conducive to hardware implementation using BCD algorithms. However, based on our investigation on the performance requirements for decimal processing in existing applications[1], potential benefits of hardware-implemented decimal arithmetic do not justify its cost. Since the success of a proposed standard on decimal floating arithmetic depends crucially on its adoption via actual implementation, we examine the proposed encoding's suitability for software emulation.

To elaborate, let us consider a compiler supporting a particular decimal datatype such as the 64-bit (16 decimal-digit) type. In a usage model consistent with current numerical programmers working with IEEE binary floating point numbers, one would expect the memory format to be that of the specified encoding in the standard. Hence, our goal is to have efficient software implementation of the proposed arithmetic operations taking operands and producing results in this encoding. Along this line, the current DPD-based encoding imposes some fundamental overhead for software processing. First, because one cannot perform arithmetic on DPD digits, conversions in both directions between DPD to some internal formats are inevitable. Second, the base-1000 nature of DPD encoding does not favor large radix arithmetic algorithms which can be exploited in software algorithms. Third, the segmented coefficient and exponent field imposes further inconvenience and inefficiency for software processing.

In this report, we describe a new encoding which we believe is more suitable for software implementation. The main characteristics of this encoding are the following.

1. It uses a purely binary encoding for the entire decimal coefficient c. For example, in the 16-digit case we encode the coefficient c, $0 \leq c < 10^{16}$, entirely in binary format, and hence the name Binary-Integer Decimal (or Big-Integer Decimal), BID.

[1] Applications we examined include Java-based financial workload, Java Spec benchmarks, and TPC-H benchmark. Typical decimal processing takes about 2% of the total time.

2. As in the case for IEEE binary floating point encoding, the coefficient (significand) field and the biased exponent field are stored contiguously.

3. Analogous to IEEE binary floating point encoding method, an "implicit bit" technique is used to economize on the bits required for the coefficient field. For example, one does not need 54 bits to store a coefficient c in the range $0 \leq c < 10^{16}$.

4. The encoding offers some interoperability with IEEE *binary* floating point. For a good portion of the numerical range, a decimal encoding corresponds to an IEEE binary floating point encoding of another finite number. Simple mathematical relationships exist between the two numerical values. This property allows a decimal floating pointer operation be substituted by some sequences (not necessarily unique) of binary floating point operations. Because binary floating points are often well implemented in hardware, such substitutions offer further possibilities in efficient software implementation.

In what follows, Section 2 describes encoding of numerical and non-numerical values into BID; Section 3 describes decoding of BID into numerical (and non-numerical) values; Section 4 describes some of the experiment we have done to illustrate BID's friendliness for software emulation; and Section 5 describes other related works that we are pursuing.

2 Encoding to BID

We discuss encoding and decoding separately. Encoding describes how one derives a bit pattern corresponding to a numeric or non-numeric value. The resulting bit patterns do not span all possible bit patterns; the section on decoding will discuss how one interprets all possible bit patterns. In short, one can relate encoding to that of the output produced by decimal arithmetic operations; and relate decoding to the interpretation of input operands to decimal arithmetic operations.

2.1 Encoding of Numerical Values

Consider numerical values of the form

$$(-1)^s \times 10^{E-\beta_p} \times c,$$

where

$$0 \leq E \leq 3 \times 2^{w-2} - 1, \quad 0 \leq c < 10^p, \quad \text{and} \quad p = 3J + 1.$$

The parameters w (an exponent range specifier), β_p (a bias value), and p (the number of decimal digits), are set corresponding to a particular format in question. The values s (the sign bit), E (the biased exponent), and c (the coefficient), are variables expressing the particular decimal value to be encoded. The parameters and ranges for the three fixed-precision formats we focus on in this report are given in Table 1.

The sign bit is obviously encoded as 1 binary bit with 0 signifying a positive number, and 1, negative.

The binary encoding of E, because of the range $0 \leq E \leq 3 \times 2^{w-2} - 1$, always fit in w bits where the two msbs can only be `00`, `01`, `10` and not `11`.

For a given $p = 3J + 1$, $J = 2, 5, 11$, the binary encoding of the coefficient c can be represented in $10J + 4$ bits. Moreover, because $2^{10J} > 10^{3J}$, whenever the binary encoding of c cannot fit into $10J + 3$ bits, the 4 msbs can be at most as large as `1001`. (Otherwise, the value c will be at least 10×2^{10J} which is bigger than 10^p.) In otherwords, the three msbs must always be `100` in this case.

Format	Parameters w	$p = 3J + 1$	J	$\beta_p = 3 \times 2^{w-3} + p - 2$	Ranges $E \in [0, 3 \times 2^{w-2} - 1]$	c
Dec32	8	7	2	101	$[0, 191]$	$0 \leq c < 10^7$
Dec64	10	16	5	398	$[0, 767]$	$0 \leq c < 10^{16}$
Dec128	14	34	11	6176	$[0, 12287]$	$0 \leq c < 10^{34}$

Table 1: Parameters and Ranges for the 3 Formats

Using these properties, the entire encoding is designed as follows. When the coefficient c fits into $10J + 3$ bits, we encode the value $(-1)^s \times 10^{E-\beta_p} \times c$ into $w + 10J + 4$ (which is 32, 64, or 128) bits as in Figure 1.

$\leftarrow$ 1bit $\rightarrow$	$\leftarrow$ w bits $\rightarrow$	$\longleftarrow$ $10J + 3$ bits $\longrightarrow$
s	binary_encode(E)	binary_encode(c)

Figure 1: Encoding When Binary(c) Fits in $10J + 3$ Bits

When c does not fit into $10J + 3$ bits, then as noted before its three msbs must always be `100`. Hence, similar to IEEE binary floating point, we do not store these bits explicitly; only the $10J + 1$ least significant bits of c need to be stored. Of course, certain indicators must exist to signify that this is the case. We therefore encode the value $(-1)^s \times 10^{E-\beta_p} \times c$ into $w + 10J + 4$ (which is 32, 64, or 128) bits as in Figure 2.

$\leftarrow$ 3bits $\rightarrow$		$\leftarrow$ w bits $\rightarrow$	$\longleftarrow$ $10J + 1$ bits $\longrightarrow$
s	`11`	binary_encode(E)	$10J + 1$ lsbs of binary_encode(c)

Figure 2: Encoding When Binary(c) Requires $10J + 4$ Bits

As noted before, the 2 msbs of binary_encode(E) are never `11`. Hence the two bits immediately following the sign bit distinguish between the two encoding cases. Also note that the case of zero is covered as c fits into $10J + 3$ bits.

2.2 Encoding Exceptional Values

The encoding of numerical values in the previous section never results in the bit pattern of `1111` immediately following the sign bit. We exploit this property to encode the exceptional values of $\pm\infty$ and NaN. The encoding of exceptional values are given in Figure 3.

3 Decoding from BID

Based on Figure 3, the exceptional values can be decoded by first recognizing the four bits following the sign bit being `1111`.

← 3bit →	← w bits →	⟵ $10J+1$ bits ⟶	
s 11	110 0.....0	00... ...00	canonical infinities
s 11	110 any	any	infinities
s 11	1110 0.....0	payloads	canonical qNaN
s 11	1110 any	any	qNaN
s 11	1111 0.....0	payloads	canonical sNaN
s 11	1111 any	any	sNaN

Figure 3: Encoding of Exceptional Values

Otherwise, we decode the bit pattern by first extracting several components. Let the bit pattern be

$$\boxed{\mathtt{b}_{L-1}\,\mathtt{b}_{L-2}\,\mathtt{b}_{L-3}\,\cdots\,\mathtt{b}_2\,\mathtt{b}_1\,\mathtt{b}_0}$$

where $L = 32, 64$, or 128. The sign bit is simply $\mathtt{b}_{L-1}$. We use the parameters w and J according to Table 1. The definition of E is

$$E = \begin{cases} \text{binary\_decode}(\mathtt{b}_{L-2}\,\mathtt{b}_{L-3}\,\cdots\,\mathtt{b}_{L-w-1}) & \text{if } (\mathtt{b}_{L-2}\mathtt{b}_{L-3}) \neq (\mathtt{11}) \\ \text{binary\_decode}(\mathtt{b}_{L-4}\,\mathtt{b}_{L-5}\,\cdots\,\mathtt{b}_{L-w-3}) & \text{if } (\mathtt{b}_{L-2}\mathtt{b}_{L-3}) = (\mathtt{11}) \end{cases}$$

We then define an integer value y again depending the case whether $\mathtt{b}_{L-2}\mathtt{b}_{L-3}$ are 11 or not.

$$y = \begin{cases} \text{binary\_decode}(\mathtt{b}_{10J+2}\,\mathtt{b}_{10J+1}\,\cdots\,\mathtt{b}_0) & \text{if } (\mathtt{b}_{L-2}\mathtt{b}_{L-3}) \neq (\mathtt{11}) \\ \text{binary\_decode}(\mathtt{1\ 0\ 0}\,\mathtt{b}_{10J}\,\mathtt{b}_{10J-1}\,\cdots\,\mathtt{b}_0) & \text{if } (\mathtt{b}_{L-2}\mathtt{b}_{L-3}) = (\mathtt{11}) \end{cases}$$

Using y, we define the decimal coefficient c by

$$c \stackrel{\text{def}}{=} \begin{cases} y & \text{if } 0 \leq y < 10^p, \\ 0 & \text{otherwise.} \end{cases}$$

With the values s, E and c defined, the decoded numeric values is simply $(-1)^s \times 10^{E-\beta_p} \times c$. The definition of c when y is 10^p or bigger is somewhat arbitrary. We pick zero instead of other reasonable choices such as 1 or $10^p - 1$ so as to make processing of this case trivial.

In summary, this decoding specification yields for all possible bit patterns a numeric or non-numeric value.

4 Software

We illustrate in this section that BID is well suited for software implementation. Its superior performance can motivate actual implementations and support by compilers, and thus enhance the probability of success of the proposed decimal standard.

4.1 Performance of BID in Software

Crucial to the success of the proposed decimal standard is actual implementations by different vendors. All indication is that implementation in software, if at all, will be the norm.

Generally speaking, software implementation has the advantage of being flexible. In software implementation, it is natural to optimize for common scenarios. Moreover, software implementation can be customized to exploit special architectural feature of the hardware platform in question.

To illustrate that software implementation can be competitive in performance to hardware implementation, we use the TELCO benchmark designed by Cowlishaw. This benchmark aims at showing the need for performance in decimal processing in finanical calculations that involve actual monetary transactions. The TELCO benchmark simulates a telecommunication billing where billing information of telephone calls are read from a file and the final charges are tallied up. Various legal requirements on conversion rate calculations mandate that the decimal arithmetic be done correctly and thus must be simulated on machines that only have binary arithmetic. The TELCO benchmark uses Cowlishaw's software package called decNumber to perform the decimal processing part. The decNumber package is flexible and implemented with care to be reasonably efficient while being portable. In this set up, Cowlishaw observes that somewhere between 70% to 90% of the total processing time of TELCO is spent in decimal processing.

We took a close look at TELCO as well as its decimal processing engine, decNumber. First, we saw that TELCO's work load is dominated by decimal processing, consistent with what TELCO is designed to illustrate. Compiling and running TELCO, as is, on a 1.5GHz Itanium2 system, we observed that roughly 78% of total time is consumed by decNumber[2]. Table 2 contains the details.

Time spent in decNumber	1.22 sec.
Total time	1.57 sec.
% time spent in decimal	78%

Table 2: Running TELCO as is on an 1.5GHz Itanium2 system

TELCO's decimal processing is encapsulated in decNumber, which defines an abstract decimal datatype with a number of supported operations such as addition, multiplication, rescaling (effecting explicit rounding), and conversion to and from character strings. The internal datatype uses integers to represent the decimal floating point exponent as well as base-10,000 digits. The TELCO benchmark's data file uses character strings for the decimal data input, which are converted to the internal representation once on input by a decNumber function. After intense decimal processing, TELCO prints an output file of decimal numbers in character strings.

We now consider the scenario when BID is the encoding for the 16-digit decimal type proposed in the standard, and that the decNumber package uses this as the native type, meaning that input and output operands are encoded in this format. Leaving the API of decNumber unchanged, we put in our software emulation for the operations supported by decNumber. We ran Telco again on the same input content, except that the numbers are now in BID encoding instead of the original character strings. The output of the TELCO is still a file of character strings. The details of this experiment is tabulated in Table 3.

TELCO aims at capturing typical decimal computations needed in real-life financial transactions. We illustrated here that our software performance for such calculations are comparable to that of dedicated hardware. More important is that the demonstrated performance potential can encourage different vendors to support the proposed standard via actual implementations.

What contributes to the reported good performance? The BID encoding is in fact a crucial factor, and we will explain this now.

[2]In a previous experiment on a 1GHz system we reported on, about 83% of total time was spent in decimal. We attribute this difference to a better gnu compiler for the underlying decimal processing part.

	TELCO in BID		speedup	TELCO as is	
TOTAL BENCHMARK TIME	0.40 sec		3.9x	1.57 sec	
Decimal Time	0.15 sec	(38%)	8.1x	1.22 sec	(78%)
average latency (cycles)					
add	12	(4 million)			
mul	12	(2.5 million)			
rescale	40	(2.5 million)			

Table 3: TELCO with BID as the Native Encoding on 1.5GHz Itanium2

4.2 Crucial Properties of BID

Doing arithmetic on operands encoded in some compact format must involve unpacking and packing. This is true regardless if the arithmetic is to be implemented in hardware or software, or the underlying arithmetic is binary or decimal floating point. One must separate components that are packed together compactly but yet have to be processed differently, such as the exponents and the significands.

The unpacking eventually results in representations of the respective components that allow arithmetic to be carried out. On most current computers, binary arithmetic, integer or floating-point, are superbly implemented with low latency and/or high throughput. BID encoding is designed so that unpacking it into binary representations of the decimal coefficient and biased exponent are extremely efficient. Packing components already in binary format back to BID is also efficient.

It may perhaps be a misconception that one must resort to digit-by-digit and integer-based algorithm in order to emulate decimal arithmetic correctly on binary hardware. Efficient algorithms in fact exist that can use binary integer as well as floating-point arithmetic. For example, consider right shifting 2 digits followed by rounding downwards a decimal integer coefficient c represented in binary floating point. If suffices to compute correctly $\lfloor c/10^2 \rfloor$. Mutiplication $d \leftarrow \lfloor c \times 10^{-2} \rfloor$ will yield the desired result to within a rounding error or two. The correct answer can be obtained by examining the remainder $10^2 d - c$. This basic method can be further refined with various techniques commonly used in the communities that routinely perform rounding-error analysis.

In brief, our implementation of BID first exploit the situations where minimal unpacking and packing suffices. These situations sometimes allow us to almost avoid all unpacking and packing. For example multiplication of moderate-length coefficients can be simply carried out by integer product on the two input operands with their top bits masked off. Only when such short cuts cannot handle the situation at hand then we fully unpack the operands into an "internal representation" IR. In our case, IR is an integer representing the exponent and a floating-point number containing the coefficient c. IEEE binary floating-point algorithms are then used to carry out the arithmetic.

In contrast, because densely packed decimal, DPD, is not a format in which one can carry out arithmetic, full-scale unpacking and packing are inevitable. Moreover, the base-1000 nature of DPD either restricts one to digit-by-digit (base-1000) algorithms, or requires further conversions between base-1000 and a higher base.

We carried out the following additional experiments based on TELCO. First, we use as the native type the current DPD-based encoding in the 754R draft, dated June 23, 2005. The input data file is translated to DPD encoding. The decNumber arithmetic is based on conversion of DPD inputs into IR, execution of arithmetic using our floating-point based algorithms, followed by conversion back from IR to DPD. Next, we re-run our BID-based TELCO, but this time taking no short cuts. That is, the decNumber arithmetic is based on conversion of BID inputs into IR always, execution of

arithmetic using our floating-point based algorithms, followed by conversion back from IR to BID. We tabulate the results in Table 4. We also include the previous data in the two last columns for easy reference.

	TELCO			
	DPD	BID (slow)	BID (fast)	as is
TOTAL BENCHMARK TIME	1.01 sec	0.76 sec	0.40 sec	1.57 sec
Decimal Time	0.65 sec	0.42 sec	0.15 sec	1.22 sec
average latency (cycles)				
add	88	61	12	
mul	95	68	12	
rescale	101	74	40	

Table 4: TELCO in DPD and BID

We make several remarks at this point.

1. The fact that BID allows for short cuts is a crucial factor in its outstanding performance. This nice property is absent in DPD based encoding, and only present in a very limited sense for a higher-base BCD encoding, such as millenial-digit based encoding.

2. Binary-arithmetic based algorithms are in fact suitable for the implementation of decimal arithmetic. Despite the additional cost of conversions between DPD and IR, the overall TELCO benchmark still reaps a 33% or so gain in overall performance, and an almost two-fold increase in decimal processing performance.

3. Based on the previous observation, a viable approach to hardware implementation of decimal arithmetic is to realize binary-arithmetic based algorithms on binary arithmetic hardware. This is cost effective because for the foreseeable future, highly optimized and sophisticated binary hardware will be present and available for re-use.

5 Future Work

On software implementation of IEEE 754R arithmetic in BID encoding, we continue to explore good implementation on both the Itanium and IA32 architectures. This includes identifying practical common cases that allow realization of arithmetic without extensive conversions (unpacking/packing). When unpacking/packing becomes necessary, we will continue to explore good internal representation formats for each of the architectures. Extending current work to IA32 and then to the 128-bit datatype are our immediate goals.

In parallel, we will also explore hardware implementation method for decimal arithmetic in BID encoding. The binary encoding is not as incompatible to decimal arithmetic as it seem. Afterall, the efficient software implementation is based on binary floating-point arithmetic. Since basic rounding in decimal can be accomplished by truncation to integer after multiplication of a suitable binary floating-point number, a fused-multiply-add type of structure can be used to implement decimal floating-point arithmetic represented in binary. It is somewhat straightforward to implement floating-point decimal addition in this way. Although in theory decimal multiplication can be similarly implemented, the data width can be rather long. We are currently investigating on methods that will reduce this data width.

Even while awaiting further results, evidence of BID's clear superiority for software implementation is strong. Its performance advantages over the current proposed encoding is significant: significant enough that should BID not be standardized, compilers implementing IEEE decimals may very well try to circumvent the standard in order to accommodate BID.

6 Acknowledgment

People contributed to this work include Marius Cornea, John Crawford, John Harrison, Jeff Kidder, Ricardo Morin, and Peter Tang, all from Intel Corporation, and Maxim Naumov from Purdue University.